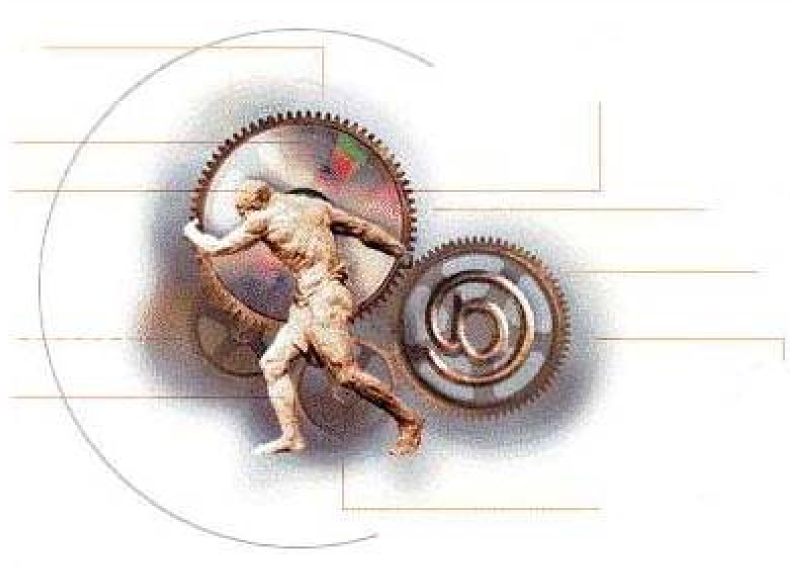
資料結構(Data Structures)

Course 2: Fundamentals of Recursion (遞迴基礎)



授課教師:陳士杰
國立聯合大學 資訊管理學系

Outlines

- 本章重點
  - Def., 與Non-recursion的比較
  - 種類
  - 相關議題
  - Recursive Function 求解

Recursion Algorithm

- 通常，一個演算法若要重覆執行某一段程式碼，會採用以下兩種程式撰寫方法之一來實作：
  - **Iteration (迴圈)**
  - **Recursion (遞迴)**
- **Def**: algorithm (或function)中含有**self-calling** (自我呼叫)的敘述存在。

遞迴的種類:

直接遞迴 (Direct Recursion):

- 函式或程序**直接呼叫本身**時稱之。

```
void Function2(void)
{
........
Function2( );
........
}
```

間接遞迴 (Indirect Recursion):

- 函式或程序先呼叫另外的函式，再從另外函式呼叫原來的函式稱之。

```
void Function1(void)
{
........
Function2( );
........
}
```

```
void Function2(void)
{
........
Function1( );
........
}
```

尾端遞迴 (Tail Recursion):

- 屬於直接遞迴的特例

程式結束的前一行

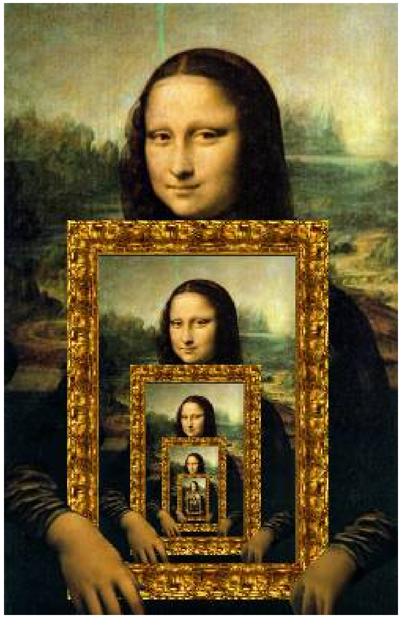
```
void Function2(void)
{
........
........
Function2( );
}
```

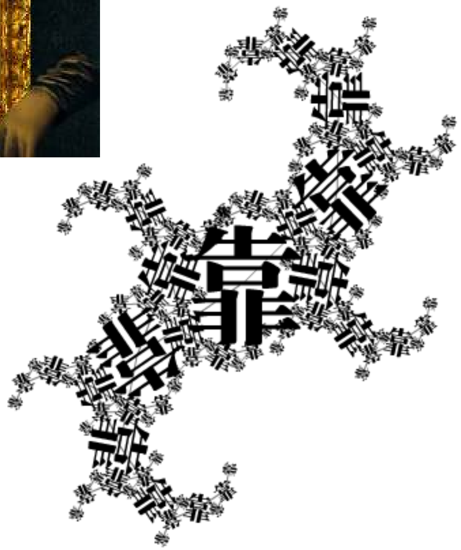
建議:用**非遞迴**方式會較有效率

- 即: 改用**迴圈** (while…, repeat…until)
- ∵遞迴要花費額外的處理 (如: stack的push, pop,…)

The Rules for Designing a Recursive Algo.

- 設計概念:遞迴演算法是採用 "逐步化簡" 的方式, 為一個大問題設計出<u>較小且性質相同的問題</u>。
  - 我們想要解決一個大小為 n 的問題, 首先是<u>把問題化簡成規模大小為 n-1 的問題, 但是解決的方法還是一樣</u> (**遞迴關係式**)。如此繼續化簡, 最後變成<u>大小為 n=1 的基本問題</u> (**終止條件**), 接著只要n=1的基本問題解決了, 原來大小為n的問題也將陸續跟著解決了。





<u>圖片來源</u>:遞迴之美: 數學, 電腦科學與碎形 (http://mmdays.com/2007/05/24/recursive/)

1. 決定基本情況 (**Base case**)
   - 此base case即為遞迴的**終止條件**
2. 決定一般情況 (**General case**)
   - 產生遞迴呼叫的指令碼，即**遞迴關係式**
3. 將上述兩種情況寫入演算法

- 遞迴的設計方法:

```
Procedure 遞迴副程式名(參數)
{
  if (Base case)
    return(結果); ....//達到終止條件時結束遞迴, 需要時回傳結果
  else
    General case; ....//利用general case執行遞迴呼叫, 需要時加上return
}
```

Recursion Algorithm常見的議題

- 遞迴常見的課題:
  - **寫一個Algorithm (or Program)**
  - **追蹤一個Recursion Algorithm (找結果、呼叫次數)**
- 遞迴議題種類:
  - **數學類**議題:
    - 階乘 (Factorial; n!)
    - 費氏數 (Fibonacci Number)
    - *兩數之最大公因數 (Greatest common divisor; G. C. D.)*
    - *二項式係數 (Binomial Coefficient; 或稱組合問題 "Combination of n objects")*
    - *Ackerman's Function*
  - **資料結構類**議題 (往後各章)
  - **其它類**議題:
    - Tower of Hanoi (河內塔問題)
    - *Permutation (排列問題)*

階乘 (Factorial; n!)

- 階乘(n!) = 1 × 2 × 3 × … ×(n-1) × n

$$\text{Factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1 & \text{if } n > 0 \end{cases}$$

We can define

Base Case

$$\text{Factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times (\text{Factorial}(n-1)) & \text{if } n > 0 \end{cases}$$

General Case

Input: 所要求算的階乘數值n

Output: n! 結果回傳

```
Procedure Factorial(int n)
begin
    if (n = 0)
        return 1;
    else
        return (n× Factorial(n-1));
end
```

Ex: Factorial(3) = ? 呼叫了幾次函數？

Factorial(3) = 3 × Factorial(2) ⇨ Factorial(3) = 3 × 2 = 6

Factorial(2) = 2 × Factorial(1) ⇨ Factorial(2) = 2 × 1 = 2

Factorial(1) = 1 × Factorial(0) ⇨ Factorial(1) = 1 × 1 = 1

Factorial(0) = 1

Return

Return

Return

⇨呼叫了**4次** (含Factorial(3))，計算結果為**6**

※ Factorial(n)會被呼叫幾次? ⇨呼叫**n+1次**

Write a program in C++

```
int Factorial(int n)
{
 if (n==0)
     return (1);
 else
     return (n*Factorial(n-1));
}
```

Iterative Factorial Algorithm

Input: 所要求算的階乘數值n

Output: n! 結果回傳

```
Procedure Factorial(int n)
begin
    i = 1;
    factN = 1;
    loop (i ≤ n)
        factN = factN * i;
        i = i + 1;
    end loop
    return factN;
end
```

費氏數 (Fibonacci Number)

- Ex:

n	0	1	2	3	4	5	6	7	8	9	10
F_n	0	1	1	2	3	5	8	13	21	34	55

- 觀念:

$$F_0 + F_1 \Rightarrow F_2$$
$$F_1 + F_2 \Rightarrow F_3$$
$$F_2 + F_3 \Rightarrow F_4$$
$$F_3 + F_4 \Rightarrow F_5$$

$\Rightarrow F_a + F_b \Rightarrow F_c$

- 定義遞迴演算法:

$$F_n = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ F_{n-1} + F_{n-2}, & \text{if } n \geq 2 \end{cases}$$

Base Case

General Case

Recursive Fibonacci Algorithm

Input: 費氏數值num

Output: 回傳Fibonacci number結果

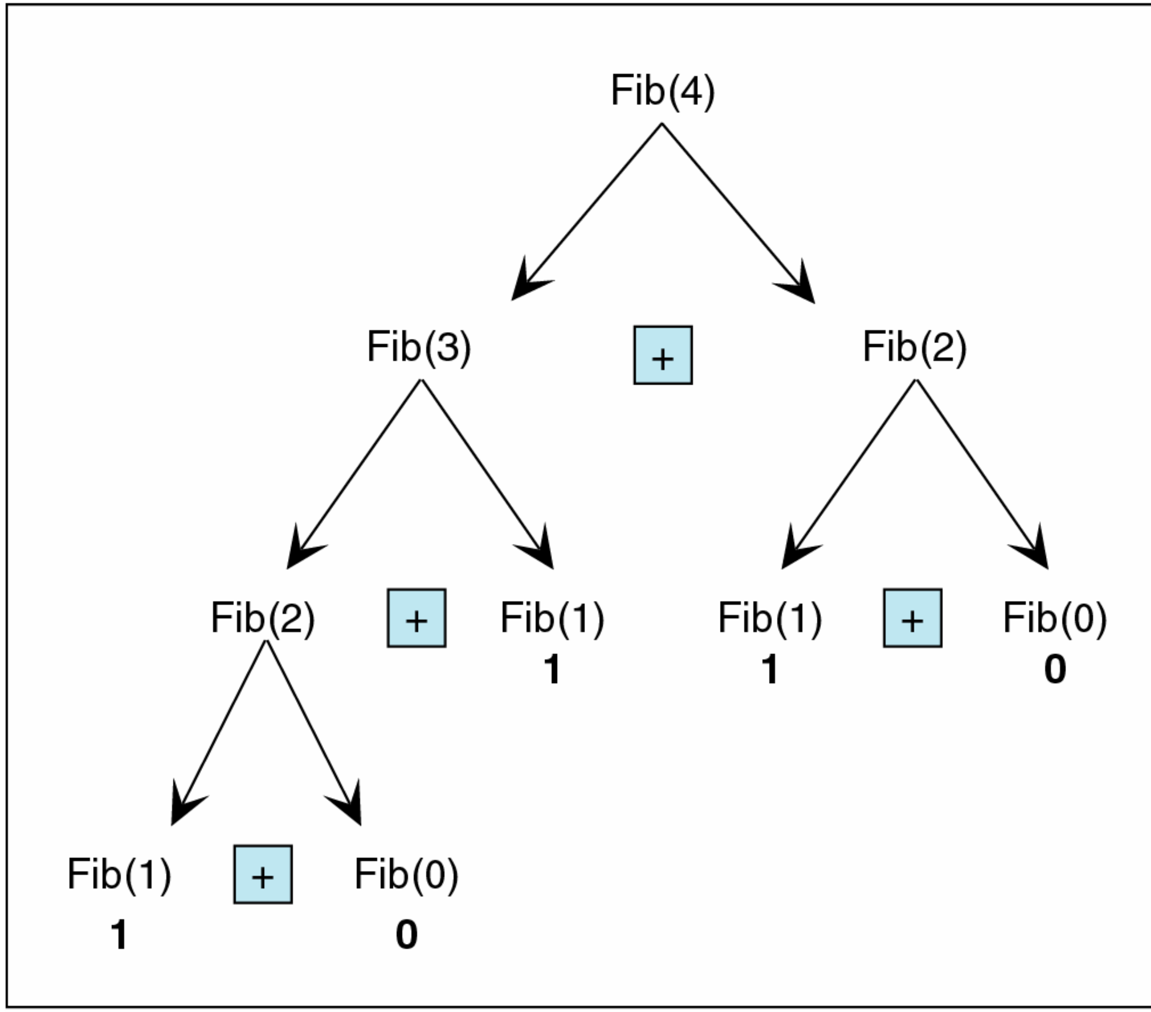
```
Procedure Fib(int num)
begin
    if (num is 0 OR num is 1)   //Base Case
        return num;
    else
        return (Fib(num-1) + Fib(num-2));
end
```

Write a program in C++

```
int Fib(int n)
{
 if (n ≤ 1)
    return (n);
 else
    return (Fib(n-1) + Fib(n-2));
}
```

- 求Fib (4) 共呼叫此遞迴函數幾次? (含Fib(4))

➪ Ans: **9次**



(b) Fib(4)

Iterative Fibonacci Number Algorithm

Input: 費氏數值num

Output: 回傳Fibonacci number結果

```
Procedure Fib(int n)
begin
    if (num is 0 OR num is 1)
        return num;
    else begin
        Fa = 0, Fb = 1;
        for(i = 2 to n)
            Fc = Fa + Fb;
            Fa = Fb;
            Fb = Fc;
        end for;
        return Fc;
    end if;
end
```

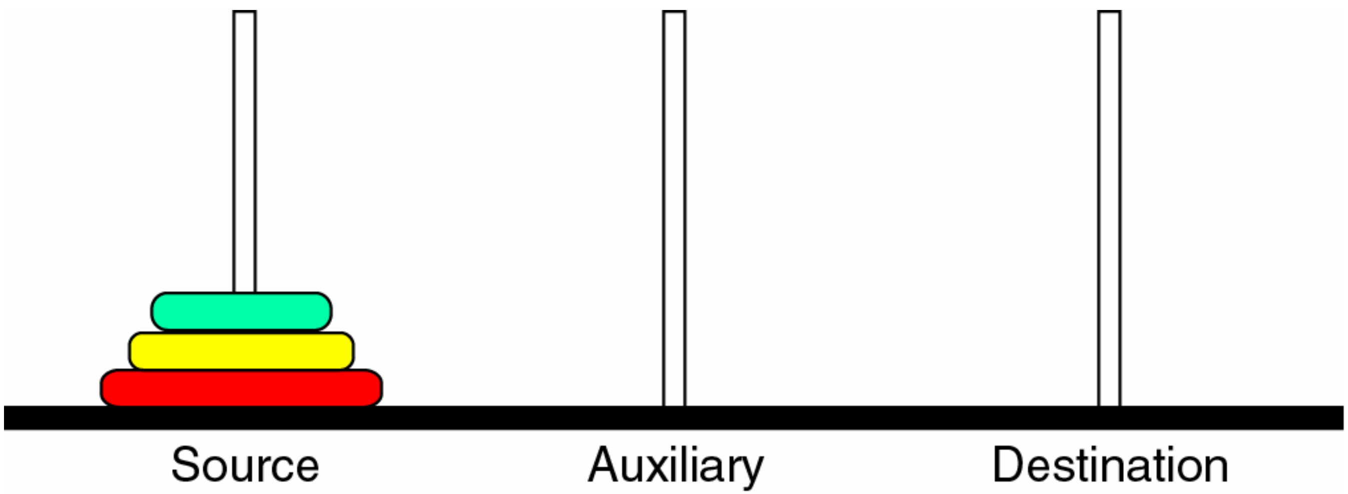
C++ Program:

```
int Fib(int n)
{
   if (n <= 1)
       return n;
   else {
         int Fa=0, Fb=1, Fc, i;
         for(i=2; i<=n; i++)
            {
              Fc = Fa + Fb;
              Fa = Fb;
              Fb = Fc;
            };
         return Fc;
       };
}
```

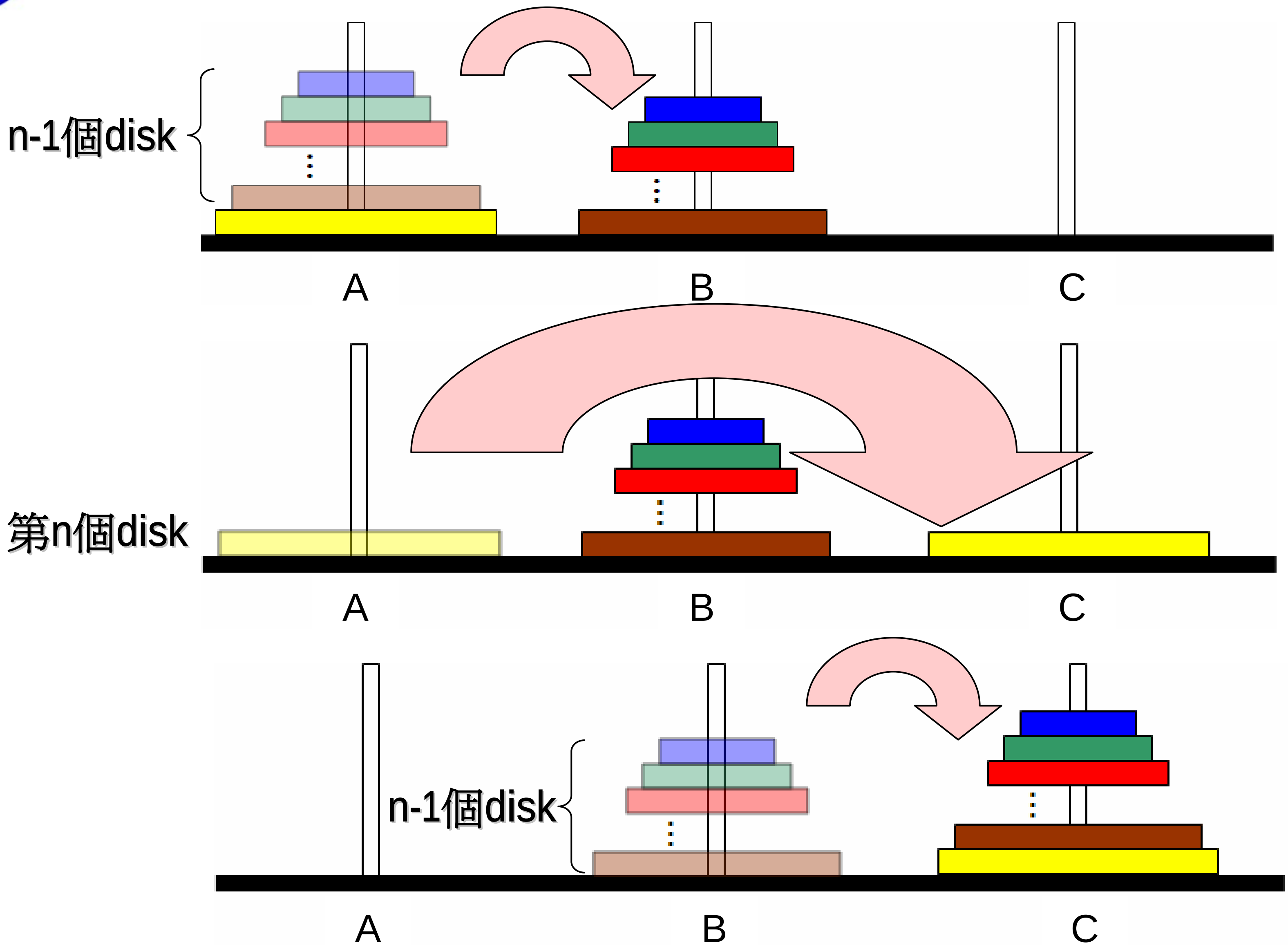
Tower of Hanoi (河內塔)

- 根據傳説，在一座據稱是在宇宙中心的古印度神廟中的僧侶們，一直想知道世界末日是在何時會發生。因此，天神指示在廟宇中**插上三根長木樁**，並在左邊的一根木樁上，**從上至下放置64片直徑由小至大的圓環形金屬盤**。將64片的金屬盤以**每天移動一片**的方式，移至三根木樁中的右邊那一根上。僧侶們若能將64片的金屬盤依規則從指定的木樁上全部移動至另一根木樁上，那麼，世界末日即隨之來到，世間的一切終將被毀滅，萬物都將至極樂世界。
- 規定：
  - 在每次的移動中，**只能搬移一片金屬盤**，
  - 過程中必須保持金屬盤是**直徑較小的被放在上層**。

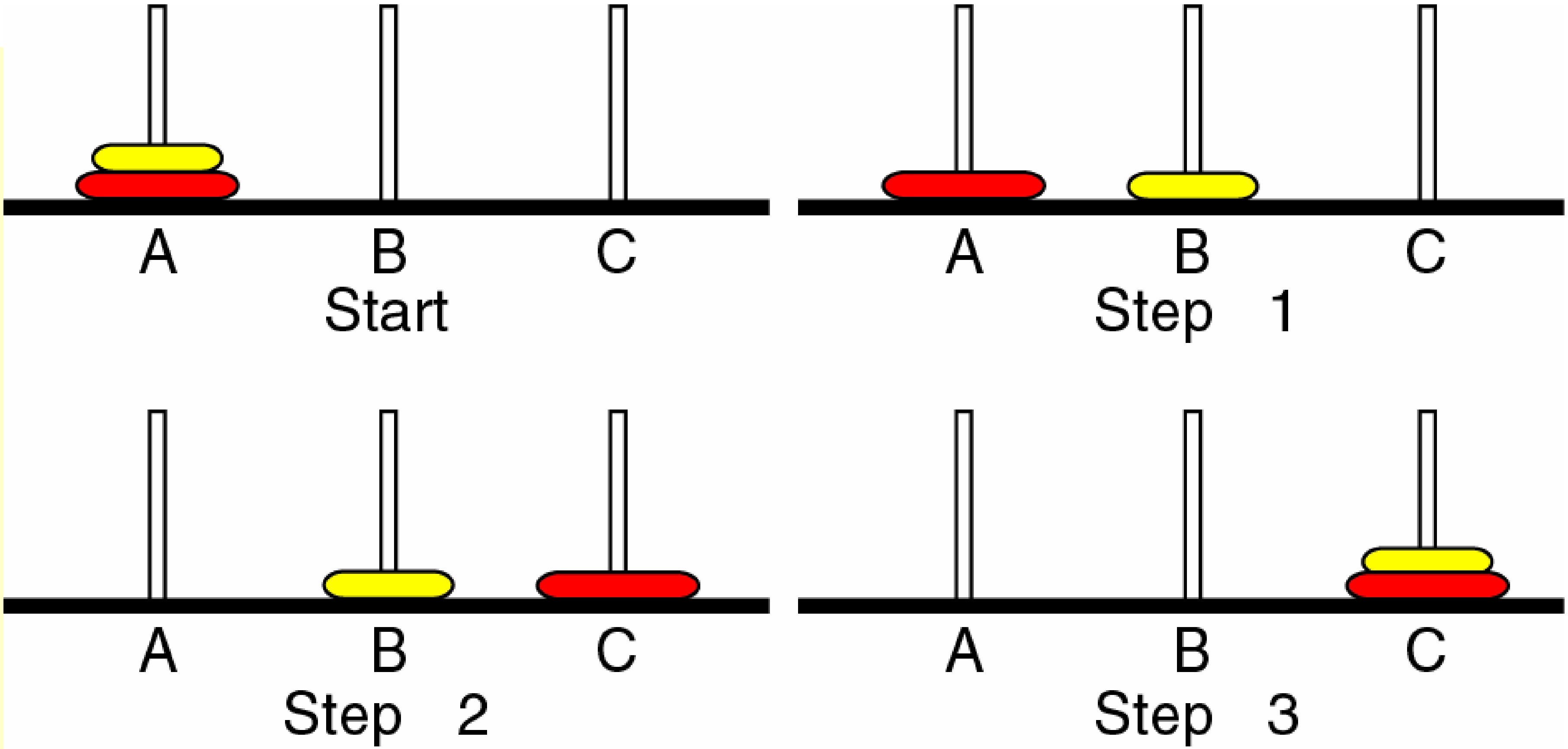
- 河內塔的規則:
  1. **一次僅能移動一個金屬盤。**
  2. **不論何時，較大的金屬盤一定要在較小的金屬盤下方。**
  3. **在移動過程中，會有一根木樁是輔助之用，於盤子搬移時暫存尚未到達目的地之盤子。**
- Need to have **2^{64} – 1 moves** to do this task.



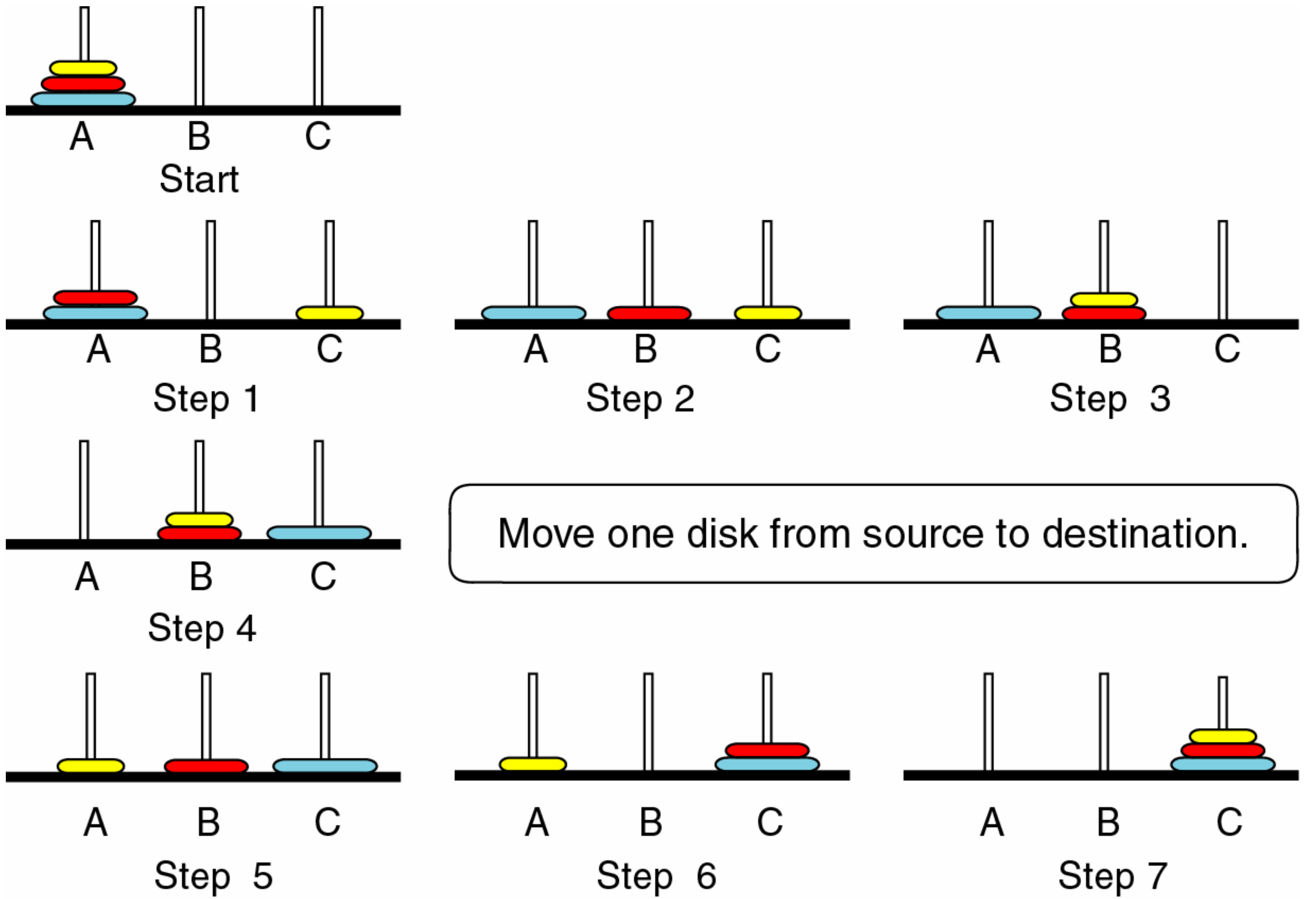
河內塔問題的解題觀念



以2個disk為例



以3個disk為例


A B C
Start
A B C
Step 1
A B C
Step 2
A B C
Step 3
A B C
Step 4
Move one disk from source to destination.
A B C
Step 5
A B C
Step 6
A B C
Step 7

- 根據上述範例，我們可以設計出河內塔問題的遞回定義：
  - Base case: (僅有<u>一個盤子</u>時)
    - Move one disk from A to C
  - General case: (盤子個數<u>多於一個</u>時)
    - Move n-1 disks from A to B
    - Move one disk from A to C
    - Move n-1 disks from B to C

Recursive Tower of Hanoi Algorithm

Input: n個盤子，三根柱子(A, B, C)

Output: 移動過程

```
                        (來源)       (目的)      (輔助)
Procedure Towers(int n, char A, char C, char B)
begin
    if (n = 1)   //Base Case
        print("Move disk", n, "from", A, "to", C);
    else begin
            Towers(n-1, A, B, C);
            print("Move disk", n, "from", A, "to", C);
            Towers(n-1, B, C, A)
        end;
end
```

Towers(3,A,C,B)

※共呼叫了**7次**Hanoi的演算法
(含主程式所呼叫的那一次)

- ➪Disk 1: A → C
- ➪Disk 2: A → B
- ➪Disk 1: C → B
- ➪Disk 3: A → C
- ➪Disk 1: B → A
- ➪Disk 2: B → C
- ➪Disk 1: A → C

Input: n個盤子，三根柱子(A, B, C)

Output: 移動過程

(來源) (目的) (輔助)

```
Procedure Towers(int n, char A, char C, char B)
begin
    if (n > 0)
        Towers(n-1, A, B, C);
        print("Move disk", n, "from", A, "to", C);
        Towers(n-1, B, C, A)
end
```

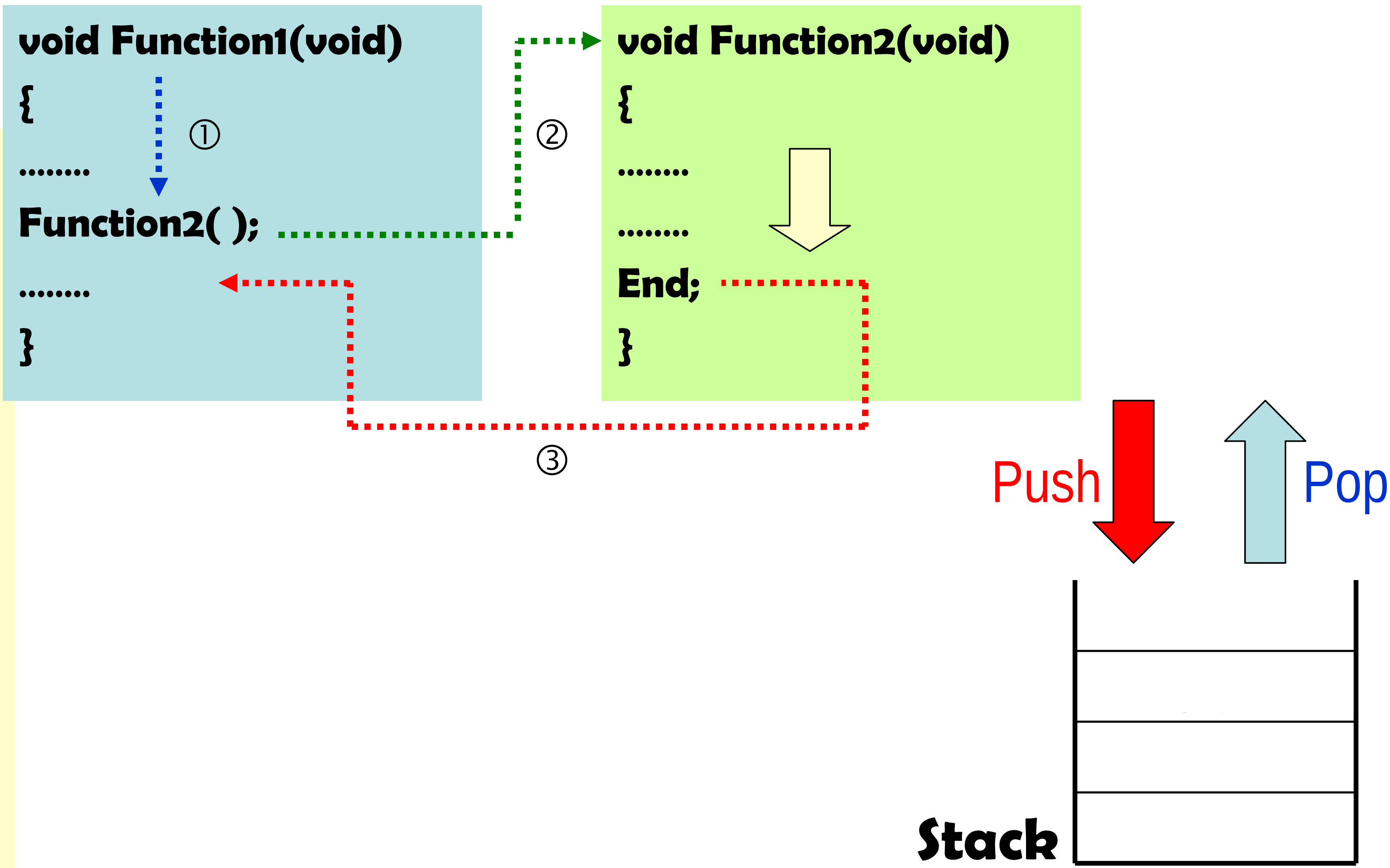
Summary of the Recursion

Note:

當以下的問題有任何一個為 no，則你不該使用遞迴來設計演算法:

1. 問題的演算法或資料結構本身是否就具備**遞迴的特性**?
2. 使用遞迴求解問題是否能**更簡化**或是**更易了解**?
3. 使用遞迴求解問題是否能在**可接受的時間**或**記憶體空間限制**下完成?

How Recursion Works



① 要**保存Function1當時執行的狀況**，即Push下列資料到Stack中。

- **參數值 (Parameter)**
- **區域/暫存變數值 (Local Variable)**
- **返回位址 (Return Address)**

② 要做**控制權轉移** (Jump to Function2)

③ Recursion動作結束時，要**Pop Stack**，以取出參數、區域/暫存變數值及返回位址，then goto "Return Address"。

⇨Push, Jump, Pop皆**耗時**，∴**效率差**

- Recursion與Non-recursion的程式可以**互相改寫**!!

Recursion 與 Non-recursion 的比較

	Recursion	Non-recursion	
優			缺
缺			優
	–		

補　充

每一個費氏數的表示方式

- 每一個費氏數皆有3種表示方式，例如:
  - $F7 = F5 + F6$

    $= F8 – F6$

    $= F9 – F8$
  - $F99 = F98 + F97$

    $= F100 – F98$

    $= F101 – F100$

最大公因數 (Greatest Common Divisor; G. C. D.)

- The greatest common divisor (gcd) of two integers can be found using Euclid's algorithm (歐幾里德演算法; 即輾轉相除法) as follows:

$$\gcd(A,B) = \begin{cases} B, & \text{if } (A \bmod B) = 0 \\ \gcd(B, A \bmod B), & \text{otherwise} \end{cases}$$

Base Case

General Case

- Calculates the greatest common divisor gcd(A,B).

Input: 輸入二個整數A與B

Output: 傳回A與B的最大公因數

```
Procedure gcd(int A, int B)
begin
    if (A mod B = 0)   //Base Case
        return B;
    else
        return gcd(B, A mod B);
end
```

Write a program in C++

```
int gcd(int A, int B)
{
 if ((A%B)==0)
    return (B);
 else
    return (gcd(B, A%B));
}
```

範例說明

- Ex 1: gcd(48, 18)

Sol:

gcd(48, 18)

gcd(18, 12)

gcd(12, 6)

Return 6

⇨共呼叫了**3次**，結果爲**6**

- Ex 2: gcd(18, 48)

Sol:

gcd(18, 48)

前後轉換

gcd(48, 18)

gcd(18, 12)

gcd(12, 6)

Return 6

⇨共呼叫了**4次**，結果爲**6**

■二項式係數 Binomial Coefficient

- 或稱**組合問題** (Combination of n objects)

$$C_m^n = \binom{n}{m} = \frac{n!}{m!(n-m)!}$$

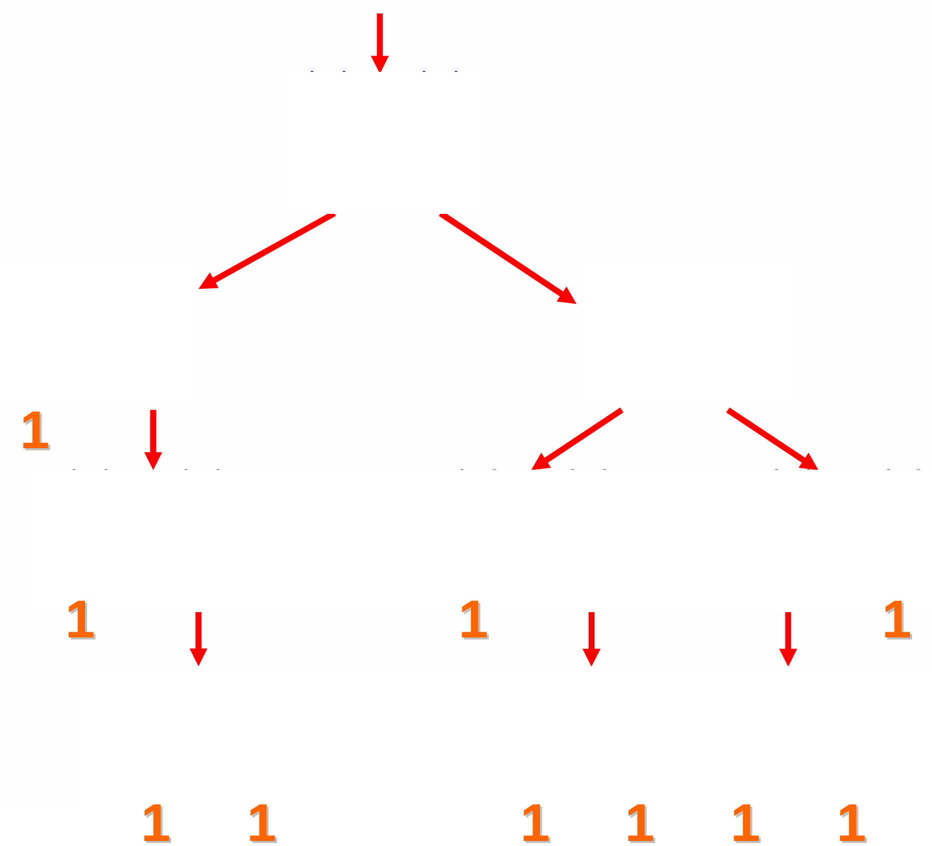
- Recursive Algorithm Definition

Base Case

$$C_m^n = \begin{cases} 1, & \text{if } n = m \text{ 或 } m = 0 \\ \binom{n-1}{m} + \binom{n-1}{m-1}, & \text{otherwise} \end{cases}$$

General Case

- Find the result of Bin(5,3) and how many invocation of Bin? (including Bin(5,3))



⇨呼叫 **19次**

⇨結果爲 **10**

- Calculates the binomial coefficient Bin(n,m).

Input: 輸入二項式的n與m

Output: 傳回二項式計算結果

```
Procedure Bin(int n, int m)
begin
    if (n=m or m=0)   //Base Case
        return 1;
    else
        return (Bin(n-1, m) + Bin(n-1, m-1));
end
```

```
int Bin(int n, int m)
{
 if (n ==0 | | n==m)
     return (1);
 else
     return (Bin(n-1, m) + Bin(n-1, m-1));
}
```

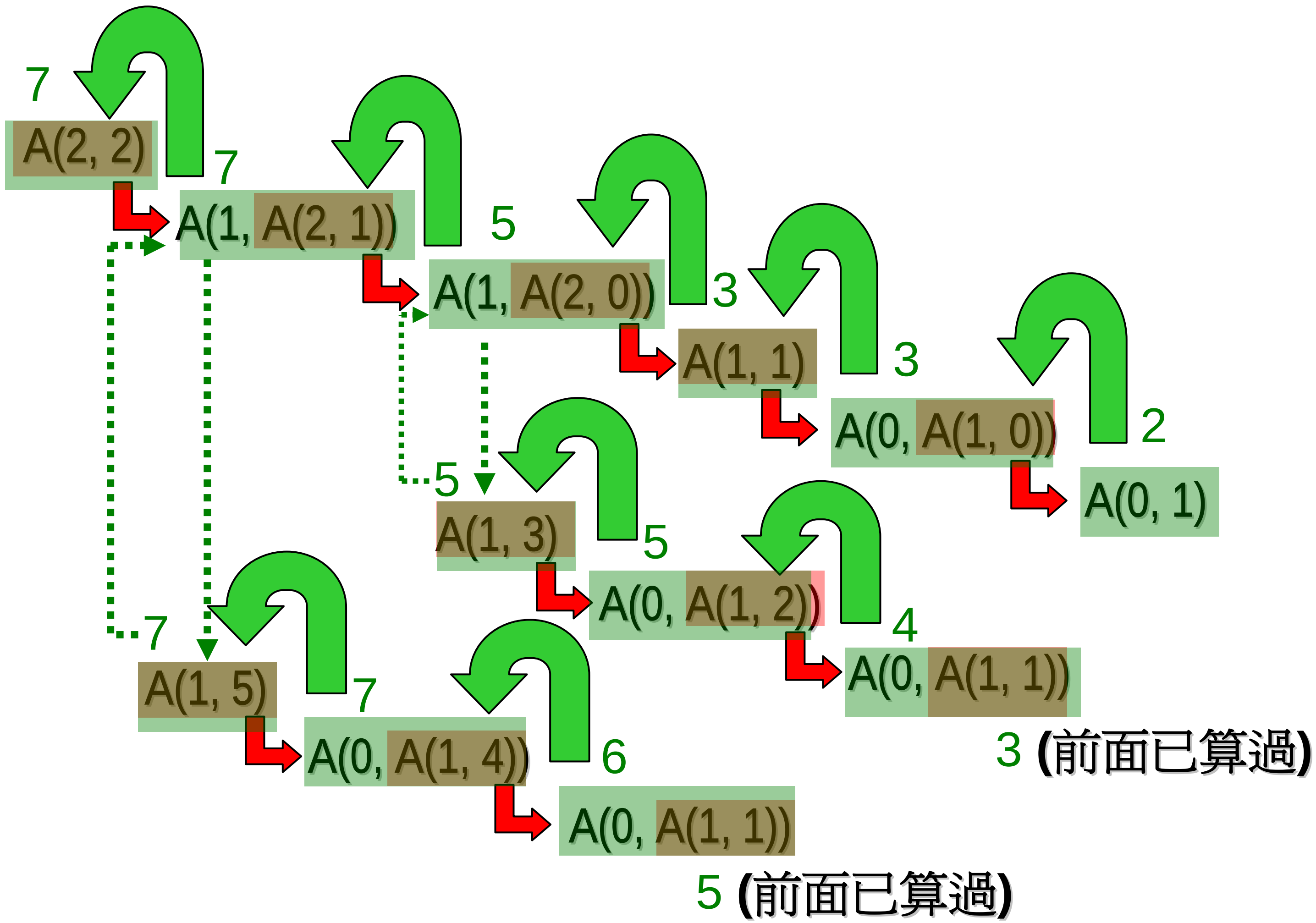
Recursive Algorithm Definition

$$\text{Ackerman}(M, N) = \begin{cases} N + 1 & \text{if } M = 0 \\ \text{Ackerman}(M - 1, 1) & \text{if } N = 0 \\ \text{Ackerman}(M - 1, \text{Ackerman}(M, N - 1)) & \text{otherwise} \end{cases}$$

Base Case

General Case

- 試求 ①A(2, 2)=? ②A(2, 1)=? ③A(1, 2)=?



Permutation (排列問題)

- 給予a, b, c三個字元, 請印出所有的Permutation。
  - 共有3! = 6 種排列

a b c
a c b

b a c
b c a

c a b
c b a

遞迴概念:

⇒若有n個不同的資料, 則**n個資料輪流當頭**, 剩餘資料去遞迴

$$\mathrm{Perm}(a,b,c)=\begin{cases}'a'+\mathrm{Perm}(b,c)\\'b'+\mathrm{Perm}(a,c)\\'c'+\mathrm{Perm}(a,b)\end{cases}$$

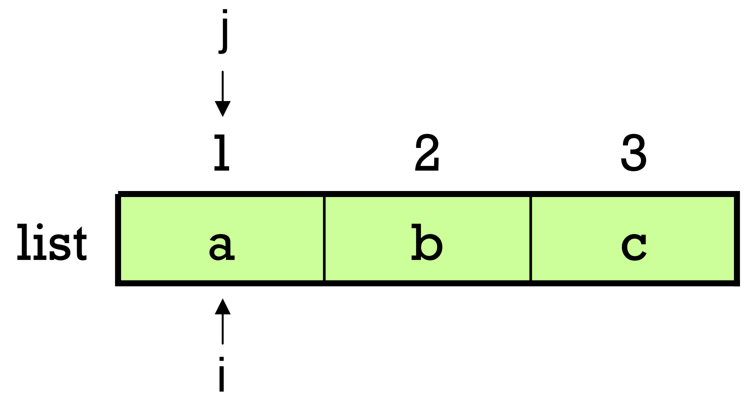
```
void perm(list[ ], int i, int n) //此函式可產生從 list[i] 至 list[n] 的元素排列
{
   if (i==n)
     {
       for (j=1; j<=n; j++)
           print(list[j]);
     }
    else
         {
           for(j=i; j<=n; j++)
               {
                   swap(list[i], list[j]);
                   perm(list[ ], i+1, n);
                   swap(list[i], list[j]);
               }
         }
}
```

前半部for迴圈

後半部for迴圈

範例：給予a, b, c三個字元，請印出所有的Permutation

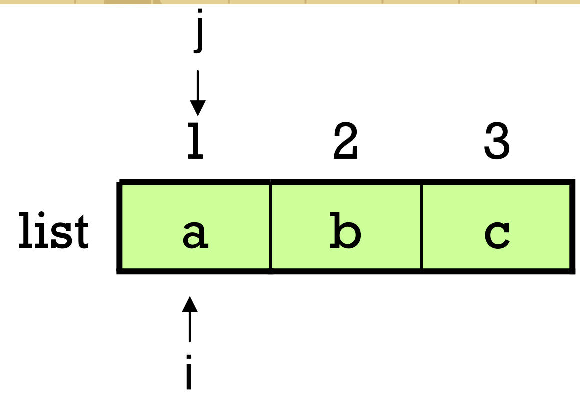
- Ans:



呼叫遞迴函式 perm(list[ ], 1, 3)

- 註：為了方便講解起見，變數j在不同的遞迴層次中，將以j’, j’’, …等示之。如：原呼叫層用j，第一層遞迴用j’，第二層遞迴用j’’…。

解題流程



- for (j=1; j<=3; j++) …//∵i≠n, ∴執行後半部的for迴圈
  - j=1時…//排出以a為首的3字元排列情況
    - swap(list[1], list[1]) …//讓a為3字元串列之首
    - perm(list[ ], 2, 3) …//處理後兩個字元
      - for(j'=2; j'<=3; j'++) …//∵i≠n，∴執行後半部的for迴圈
        - j'=2時
          - swap(list[2], list[2]) …//讓b為後二字元串列之首
          - perm(list[ ], 3, 3) …// ∵i=n, ∴執行前半部for迴圈, 印出a, b, c
          - swap(list[2], list[2]) …//還原前次的swap
        - j'=3時
          - swap(list[2], list[3]) …//讓c為後二字元串列之首
          - perm(list[ ], 3, 3) …// ∵i=n, ∴執行前半部for迴圈, 印出a, c, b
          - swap(list[2], list[3]) …//還原前次的swap
    - swap(list[1], list[1]) …//還原前次的swap

- for (j=1; j<=3; j++) …//∵i≠n，∴執行後半部的for迴圈
  - j=2時…//排出以b為首的3字元排列情況
    - swap(list[1], list[2]) …//讓b為3字元串列之首
    - perm(list[ ], 2, 3) …//處理後兩個字元
      - for(j'=2; j'<=3; j'++) …//∵i≠n，∴執行後半部的for迴圈
        - j'=2時
          - swap(list[2], list[2]) …//讓a為後二字元串列之首
          - perm(list[ ], 3, 3) …// ∵i=n, ∴執行前半部for迴圈, 印出b, a, c
          - swap(list[2], list[2]) …//還原前次的swap
        - j'=3時
          - swap(list[2], list[3]) …//讓c為後二字元串列之首
          - perm(list[ ], 3, 3) …// ∵i=n, ∴執行前半部for迴圈, 印出b, c, a
          - swap(list[2], list[3]) …//還原前次的swap
    - swap(list[1], list[2]) …//還原前次的swap

	1	2	3
list	a	b	c

j ↓ (2)

i ↑ (1)

- for (j=1; j<=3; j++) …//∵i≠n，∴執行後半部的for迴圈
  - j=3時…//排出以c為首的3字元排列情況
    - swap(list[1], list[3]) …//讓c為3字元串列之首
    - perm(list[ ], 2, 3) …//處理後兩個字元
      - for(j’=2; j’<=3; j’++) …//∵i≠n，∴執行後半部的for迴圈
        - j’=2時

          swap(list[2], list[2]) …//讓b為後二字元串列之首

          perm(list[ ], 3, 3) …// ∵i=n, ∴執行前半部for迴圈, 印出c, b, a

          swap(list[2], list[2]) …//還原前次的swap
        - j’=3時

          swap(list[2], list[3]) …//讓a為後二字元串列之首

          perm(list[ ], 3, 3) …// ∵i=n, ∴執行前半部for迴圈, 印出c, a, b

          swap(list[2], list[3]) …//還原前次的swap
    - swap(list[1], list[3]) …//還原前次的swap

j

	1	2	3
list	a	b	c

i

■遞迴演算法的時間函數

(請見演算法數位課程, 本課程不另講解)